Corso di Algoritmi e Strutture di Dati
Esercizi

Esercizio 1. E' dato un insieme X = {1, 2, ..., n} di n oggetti. L'oggetto i-esimo ha peso p[i]. I pesi sono numeri interi positivi. Disponiamo di un contenitore in grado di trasportare al massimo un peso C.
Vogliamo determinare un sottoinsieme Y ⊆ X tale che il peso complessivo degli oggetti in Y sia esattamente uguale a C.

Soluzione. Indichiamo con *S*[*i,j*], con i∈{1..n} e j∈{0..C}, un valore booleano che sarà *true* se è possibile riempire completamente un contenitore di capacità *j* usando oggetti presi nell'insieme {1..i}, *false* altrimenti. Definiamo i casi base *S*[*1, j*] come segue:

$$S[1,j]=\begin{cases} true & \text{se } j=p[1] \text{ or } j=0 \\ false & \text{altrimenti} \end{cases}$$

Definiamo il valore *S*[*i, j*] nel caso generale *i* > 1 nel modo seguente:

$$S[i,j]=\begin{cases} S[i-1,j] & \text{se } j<p[i] \\ S[i-1,j] \text{ or } S[i-1,j-p[i]] & \text{altrimenti} \end{cases}$$

Una volta calcolati i valori *S*[*i,j*], si ha che *S*[*n,C*] = *false* implica la non esistenza di Y ⊆ X tale che il peso complessivo degli oggetti in Y sia esattamente uguale a C. Se invece *S*[*n,C*] = *true* si procede a calcolare l'insieme Y richiesto come segue:

```
integer j ← C;
set of integer Y ← ∅;
for i ← n downto 2 do
    if ( j>0 && S[i,j] ≠ S[i-1,j] ) then
        Y ← Y ∪{i};
        j ← j-p[i];
    endif
endfor
if (j>0) then
    Y ← Y ∪{1};
endif
```

Esercizio 2. E' dato un insieme X = {1, 2, ..., n} di n oggetti. L'oggetto i-esimo ha peso p[i]. I pesi sono numeri interi positivi. Disponiamo di un contenitore in grado di trasportare al massimo un peso C.
Vogliamo determinare un sottoinsieme Y ⊆ X di oggetti il cui peso complessivo sia massimo possibile e minore o uguale a C (ovvero, vogliamo riempire il contenitore il più possibile).

Soluzione. Indichiamo con *M*[*i,j*], con i∈{1..n} e j∈{0..C}, un valore intero che indica il peso complessivo massimo, minore o uguale a *j*, ottenibile considerando un insieme Y ⊆ {1, .. j}. Definiamo i casi base *M*[1*, j*] come segue:

$$M[1,j]=\begin{cases} 0 & \text{se } j<p[1] \\ p[1] & \text{altrimenti} \end{cases}$$

Definiamo il valore *M*[*i, j*] nel caso generale *i* > 1 nel modo seguente:

$$M[i,j]=\begin{cases} M[i-1,j] & \text{se } j<p[i] \\ \max\{M[i-1,j],\, p[i]+M[i-1,j-p[i]]\} & \text{altrimenti} \end{cases}$$

Il peso complessivo massimo, minore o uguale a C, ottenibile considerando un insieme Y £ {1, .. n} è quindi $S[n,C]$. Per calcolare l'insieme Y che produce questo peso complessivo massimo si procede come nell'Esercizio 1, ma inizializzando $j \leftarrow S[n,C]$ invece di $j \leftarrow C$.

Esercizio 3. Disponiamo di $n \geq 1$ monete aventi valori interi positivi $c[1], \ldots c[n]$; i valori delle monete sono arbitrari, quindi non necessariamente relativi ad un sistema monetario canonico. Attenzione: un elemento $c[i]$ rappresenta il valore di una singola moneta a disposizione, non infinite monete di quel valore. Scrivere un algoritmo basato sulla programmazione dinamica per calcolare il minimo numero di monete che è necessario usare per erogare un resto esattamente pari a R, se questo è possibile.

Soluzione. Indichiamo con $N[i, j]$ il minimo numero di monete nell'insieme $\{1, \ldots i\}$ che è necessario usare per un resto pari a j; se il resto non è erogabile, $N[i, j] = +\infty$. Definiamo i casi base $N[1, j]$ come segue:

$$N[1, j] = \begin{cases} 0 & \text{se } j = 0 \\ 1 & \text{se } j = c[1] \\ +\infty & \text{altrimenti} \end{cases}$$

Definiamo il valore $N[i, j]$ nel caso generale $i > 1$ nel modo seguente:

$$N[i, j] = \begin{cases} N[i-1, j] & \text{se } j < c[i] \\ \min\{N[i-1, j], N[i-1, j-c[i]]+1\} & \text{altrimenti} \end{cases}$$

Il risultato cercato è quindi $N[n, R]$: se tale valore è un numero finito, allora è possibile erogare il resto con $N[n, R]$ monete; se tale valore è $+\infty$, allora il resto non è erogabile.

```
integer resto( integer c[1..n], integer R )
  integer N[1..n, 0..R];
  integer i, j;
  // inizializzazione prima riga
  N[1, 0] ← 0;
  for j ← 1 to R do
      if ( j = c[1] ) then
          N[1, j] ← 1;
      else
          N[1, j] ← +∞;
      endif
  endfor
  // calcolo della matrice
  for i ← 2 to n do
      for j ← 0 to R do
          if ( j < c[i] ) then
              N[i, j] ← N[i - 1, j];
          else
              N[i, j] ← MIN( N[i – 1, j], N[i - 1, j - c[i]] + 1 );
          endif
      endfor
  endfor
  return N[n, R];
```

Esempio con R = 6, c = [5, 2, 2, 2];

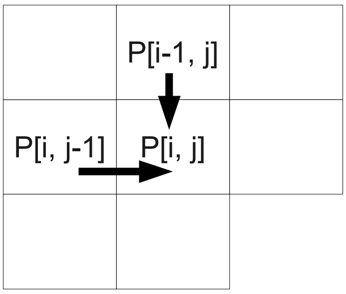
0	INF	INF	INF	INF	1	INF
0	INF	1	INF	INF	1	INF
0	INF	1	INF	2	1	INF
0	INF	1	INF	2	1	3

Esercizio 4. Si consideri una scacchiera quadrata rappresentata da una matrice $M[1..n, 1..n]$. Scopo del gioco è spostare una pedina dalla casella in alto a sinistra (1, 1) alla casella in basso a destra (n, n). Ad ogni mossa la pedina può essere spostata di una posizione verso il basso, oppure di una posizione verso destra (senza uscire dai bordi). Quindi, se la pedina si trova in (i, j) potrà essere spostata in $(i + 1, j)$ oppure $(i, j + 1)$, se possibile. Ogni casella $M[i, j]$ contiene un numero reale; man mano che la pedina si muove, il giocatore accumula il punteggio segnato sulle caselle attraversate, incluse quelle di partenza e di arrivo.
Scrivere un algoritmo efficiente che, dati in input i valori presenti nella $M[1..n, 1..n]$ restituisce il massimo punteggio che è possibile ottenere spostando la pedina dalla posizione iniziale a quella finale con le regole di cui sopra. Ad esempio, nel caso seguente:

1	3	4	-1
3	-2	-1	5
-5	9	-3	1
4	5	2	-2

l'algoritmo deve restituire 16 (le celle evidenziate indicano il percorso da far fare alla pedina per ottenere il massimo punteggio che in questo caso è 16).

Soluzione. Possiamo risolvere il problema utilizzando la programmazione dinamica. Definiamo una matrice $P[1..n, 1..n]$ di numeri reali, tale che $P[i, j]$ rappresenta il massimo punteggio che è possibile ottenere spostando la pedina dalla cella (1, 1) fino alla cella (i, j). Una volta calcolati tutti i valori degli elementi di P, la risposta al nostro problema sarà il valore contenuto in $P[n, n]$.
Sappiamo che $P[1, 1] = M[1, 1]$. Osserviamo che le celle della prima riga della matrice possono essere raggiunte esclusivamente spostando la pedina a destra, a partire dalla posizione iniziale. Quindi $P[1, j] = P[1, j - 1] + M[1, j]$, per ogni $j = 2, \ldots n$. Analogamente, le celle della prima colonna possono essere raggiunte esclusivamente spostando la pedina in basso, quindi $P[i, 1] = P[i - 1, 1]+M[i, 1]$ per ogni $i = 2,\ldots n$.
Supponiamo ora di voler calcolare $P[i, j]$ per un certo $i > 1, j > 1$. La cella di coordinate (i, j) può essere raggiunta a partire da $(i - 1, j)$, spostando la pedina verso il basso di una posizione; in tal caso avremmo che il punteggio ottenuto dopo lo spostamento è $P[i - 1, j] + M[i, j]$. Alternativamente, possiamo raggiungere la cella (i, j) a partire da $(i, j - 1)$ spostando la pedina verso destra; in tal caso il punteggio ottenuto è $P[i, j - 1] + M[i, j]$. Si faccia riferimento alla figura seguente



Chiaramente sceglieremo la mossa che ci farà ottenere il punteggio massimo, quindi possiamo scrivere, per ogni $i = 2, ..., n$ e per ogni $j = 2, ... n$:

$$P[i, j]=\max\{P[i-1, j], P[i, j-1]\}+M[i, j]$$

L'algoritmo seguente risolve il problema; in più, anche se non richiesto dal testo, l'algoritmo stampa il percorso che consente di ottenere il punteggio massimo.

```
real GiocoScacchiera( real M[1..n, 1..n] )
   real P[1..n, 1..n];
   int i, j;
   // inizializzazione
   P[1, 1] ← M[1, 1];
   for j ← 2 to n do                    // prima riga
        P[ 1, j ] ← P[ 1, j -1 ] + M[ 1, j ];
   endfor
   for i ← 2 to n do                    // prima colonna
        P[ i, 1 ] ← P[ i - 1, 1 ] + M[ i, 1 ];
   endfor
   // Calcolo della soluzione
   for i ← 2 to n do
        for j ← 2 to n do
             if ( P[ i - 1, j ] ≥ P[ i, j – 1 ] ) then
                  P[ i, j ] ← P[ i - 1, j ] + M[ i, j ];
             else
                  P[ i, j ] ← P[ i, j – 1 ] + M[ i, j ];
             endif
        endfor
   endfor
   // Stampa la sequenza di caselle visitate, a partire dall'ultima (non richiesto
dall'esercizio); si noti che NON viene stampata la casella iniziale (1, 1).
   i ← n;
   j ← n;
   while ( i > 1 and j > 1 ) do
        print "(", i, ",", j, ")";
        if ( P[ i, j ] = P[ i - 1, j ] + M[ i, j ] ) then
             j ← j - 1;
        else
             i ← i - 1;
        endif
   endwhile
   // restituisco la soluzione (infatti l'algoritmo restituisce un numero reale)
   return P[n, n];
```

Il costo dell'algoritmo è dominato dal ciclo per il calcolo degli elementi della matrice *P*, ed è pari a $\boldsymbol{\Theta(n^2)}$.